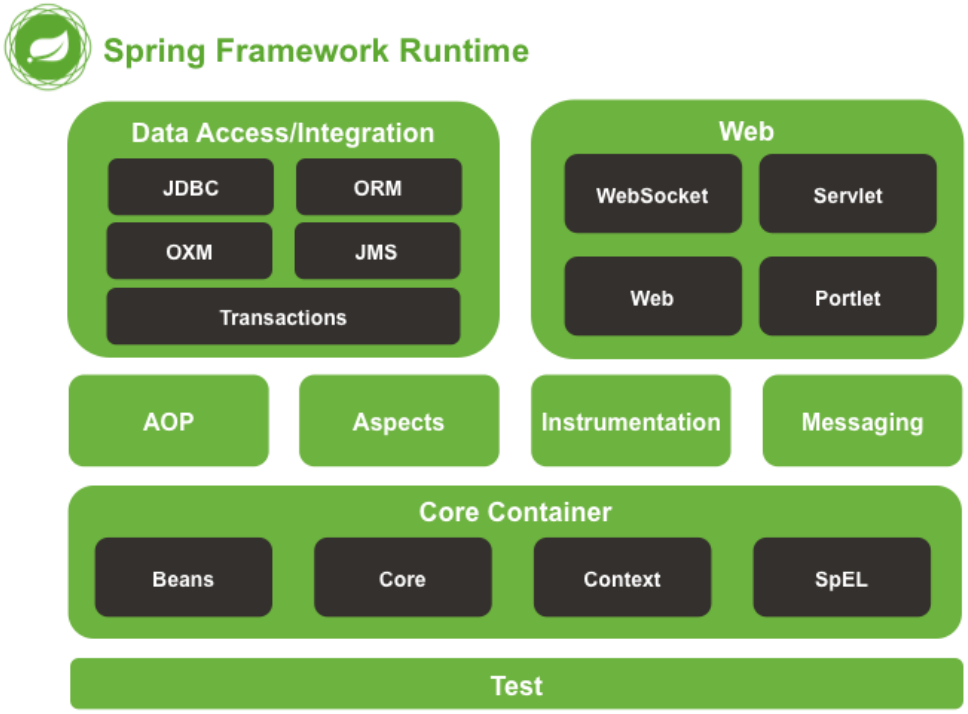
SPRING FRAMEWORKS



Capitulo 2
Spring Core Container

Contenido

1 DEPENDENCY INJECTION CONTAINER

El módulo Core es el más importante de Spring. Es el que provee el Contenedor DI. Este contenedor nos permite aplicar el patrón Dependency Injection en nuestras aplicaciones.

En forma muy resumida, el objetivo del contenedor DI es encargarse de instanciar los objetos de nuestro sistema, denominados beans, y asignarle sus dependencias. Para que el contenedor pueda llevar a cabo esta tarea, debemos, mediante información de configuración, indicarle dónde se encuentran dichos beans.

Supongamos que dentro de nuestro sistema necesitamos representar una computadora. Dicha computadora depende un microprocesador, un placa madre, memorias, disco rígido, etc. Al crear un objeto de tipo computadora tendríamos que asignarle, mediante código, estos componentes. Utilizando el contenedor de Spring, en lugar de hacer todas estas creaciones y asignaciones de dependencias que posee la computadora con sus componentes, le pediremos al contenedor que instancie un objeto de tipo computadora y el será el encargado de crear y asignar las dependencias de la computadora con sus componentes.

El principal concepto es la interfaz **BeanFactory**. Es una sofisticada implementación del patrón “Factory”, nos permite instanciar objetos de forma muy rápida y fácil, liberándonos de la necesidad de especificar las dependencias en la lógica de nuestro programa y, por lo tanto, desacoplar esta dependencia en tiempo de codificación, quedando ligada al tiempo de “inicialización / start-up” o incluso al “ejecución / run-time” ya que pueden cambiarse las implementaciones en tiempo de ejecución.

2 LOS BEANS

El contenedor DI de Spring administra uno o muchos beans. Estos beans son creados utilizando las instrucciones declaradas en la información de configuración del contenedor.

Configurar los beans mediante un archivo xml es sólo una de las formas existentes. Por ser la primera de las formas utilizadas es la que se ha elegido para implementar los ejemplos.

```
<beans>
   ...
   <bean id="miPrimerBean" class="paquete.Ejemplo"/>
   ...
</beans>
```

Internamente, Spring, guarda las definiciones de cada bean representadas por un objeto BeanDefinition. Cada BeanDefinition posee la siguiente información:

- Nombre de la clase (incluyendo el paquete) para la cual se define el bean, o el nombre de la clase factory que crea la clase.
- Información de comportamiento del bean: Si es “Prototype” o “Singlenton”, si utiliza métodos para inicialización o destrucción, etc.
- Argumentos de construcción del objeto o seteo de propiedades luego de la creación.
- Otros beans necesarios para el bean creado. (Dependencias del bean).

3 ALIASING BEANS

Para cada bean definimos un identificador unívoco para referenciarlo mediante el atributo 'id' del elemento 'bean'. Pero existe la posibilidad de referenciar al mismo bean con diferentes nombres definiéndole alias.

Definición del bean:

```
<beans>
   ...

   <bean id="miPrimerBean" class="paquete.Ejemplo"/>

   ...
</beans>
```

Definición de los alias:

```
<beans>
   ...

   <alias name="miPrimerBean" alias="miBean"/>

   <alias name=" miPrimerBean" alias="unBean" />

   ...
</beans>
```

4 BEAN SCOPES

Uno de los principales atributos que se utilizan en la definición de un bean es el atributo 'scope'. Dicho atributo permite definir el alcance de un bean. Spring posee, por defecto 4 alcances principales:

- **Singleton:** Se crea una sola instancia del bean por DI container.
- **Prototype:** Permite crear cualquier cantidad de instancias del bean.
- **Request:** Se crea automáticamente una instancia del bean por cada request. Podemos modificar el bean y solo será modificado para el request en el que nos encontramos trabajando, los otros request no verán estos cambios. Solo válido utilizando ApplicationContext para aplicaciones web.
- **Session:** Se crea automáticamente una instancia del bean por cada session. Podemos modificar el bean y solo será modificado para la session en la que nos encontramos trabajando, las otras session no verán estos cambios. Solo válido utilizando ApplicationContext para aplicaciones web.
- **Global Session:** Similar al Session, es utilizado por portlets para compartir la session entre los distintos portlets. En caso de que no estemos utilizando portlets este scope degrada al scope session. Solo válido utilizado ApplicationContext

Adicionalmente, Spring nos permite definir nuestros propios scopes.

5 INSTANCIACIÓN DE LOS BEANS

El contenedor de DI de Spring buscara en la información de configuración los datos necesarios para instanciar al bean solicitado utilizando Java Reflection.

Para indicar la clase que se deberá utilizar para instanciar el bean se utilizar el atributo 'class' del elemento 'bean'.

```
<beans>
   ...

   <bean   id="miPrimerBean"
           class="paquete.Ejemplo"/>

   ...
</beans>
```

Otra forma para indicar cómo construir un bean es utilizando un método estático factory. Para este caso, el atributo "**class**" indica el nombre de la factory para instanciar al objeto. Además, es necesario declarar otro atributo, "**factory-method**", que posee el nombre del método estático para instanciar el objeto. El tipo de objeto devuelto por la factory puede ser de cualquier tipo.

```
<beans>
   ...

   <bean   id="beanCreadoPorFactory"
           class="paquete.SoyUnaFactory"
           factory-method="creoInstancias"/>

   ...
</beans>
```

Y la última forma para construir un bean es utilizando un método factory de instancia. En este caso, no se utiliza el atributo 'class'. Se utilizaran los atributos 'factory-bean' y 'factory-method'. 'factory-bean' debe referenciar a un bean dentro del mismo contenedor (o el contenedor padre) que posee el método factory. 'factory-method' debe indicar el nombre del método a utilizar.

```
<beans>
   ...

   <bean   id="beanFactory"
           class="paquete.SoyFactory" />

   <bean   id="beanCreadoPorFactory"
           factory-bean="beanFactory"
           factory-method="creoInstancias"/>

   ...
</beans>
```

6 INSTANCIANDO EL CONTENEDOR

Ya hemos definidos los beans dentro de nuestro sistema. Ahora veamos la forma de accederlos. Spring posee dos interfaces que definen los métodos para poder utilizar nuestros beans. La primera y más simple, BeanFactory, esta interfaz define la funcionalidad básica mientras que ApplicationContext extiende la funcionalidad de BeanFactory y es la preferida al implementar aplicaciones J2EE.

```
BeanFactory beanFactory =
  new ClassPathXmlApplicationContext("/demo/beans.xml");
```

De esta forma definimos que la información de configuración de nuestros beans se encuentra en el archivo "beans.xml". Utilizando el método **getBean("nombreBean")** podemos obtener instancias de nuestros beans.

```
MiPrimerBean bean;
bean = ( MiPrimerBean ) beanFactory.getBean("miPrimerBean");
```

7 INYECCIÓN DE DEPENDENCIAS

Generalmente cuando utilizamos un objeto este no trabaja solo sino que lo hace comunicándose con otros objetos. Estos objetos (dependencias) son pasados como argumentos en constructores o seteados como propiedades (inyectados) al objeto por el contenedor DI.

Cabe destacar la importancia de la inyección de dependencias: El código se vuelve más claro y menos acoplado cuando los objetos no crean sus dependencias. (no necesitan saber dónde están localizados ni a que clase pertenecen).

Junto con la declaración del bean podemos incluir inyección de dependencias. Para lograr esto, podemos utilizar dos formas: inyección por métodos setters o inyección por constructor

7.1 Inyección por métodos setters

La inyección de dependencias se realiza a través de métodos setters luego de instanciar al objeto invocando un constructor sin parámetros o un método factory sin parámetros.

```
<beans>

  <bean id="meInyectan" class="paquete2.UnBean"/>

  <bean id="meInyectanTambien" class="paquete3.OtroBean"/>

  <bean id="beanEjemplo" class="paquete.Ejemplo">
    <property name="unBean"><ref bean="meInyectan"/></property>
    <property name="otroBean" ref="meInyectanTambien"/>
    <property name="soyUnInteger" value="1"/>
  </bean>

</beans>
```

A continuación se presenta el código de la clase Ejemplo:

```
package paquete;

public class Ejemplo {

   private UnBean unBean;
   private OtroBean otroBean;
   private int i;

   public void setUnBean( UnBean unBean ) {
      this.unBean  = unBean;
   }

   public void setOtroBean( OtroBean otroBean) {
      this.otroBean  = otroBean;
   }

   public void setSoyUnInteger(int i) {
      this.i = i;
   }

}
```

7.2 Inyección por constructor

La inyección de dependencias por medio de constructores se realizan llamando al constructor de la clase pasándole un conjunto de argumentos como dependencias (De la misma forma se realiza si el constructor fuera un método factory).

```
<beans>

   <bean id="meInyectan" class="paquete2.UnBean"/>

   <bean id="meInyectanTambien" class="paquete3.OtroBean"/>

   <bean id="beanEjemplo" class="paquete.Ejemplo" >
      <constructor-arg><ref bean="meInyectan"/></constructor-arg>
      <constructor-arg ref="meInyectanTambien"/>
      <constructor-arg type="int" value="1"/>
   </bean>

</beans>
```

A continuación se presenta el código de la clase Ejemplo:

```
package paquete;

public class Ejemplo {

   private UnBean unBean;
   private OtroBean otroBean;
   private int i;

   public Ejemplo(UnBean unBean, OtroBean otroBean, int i)  {
      this.unBean = unBean;
      this.otroBean = otroBean;
      this.i = i;
   }

}
```

Para resolver en qué posición debe ser pasado cada parámetro al constructor, el contenedor chequea el tipo del argumento. Esto puede realizarse siempre que no se encuentre una ambigüedad en los tipos. En caso de encontrar una ambigüedad, los parámetros son pasados en el orden especificado en la definición del bean.

La mejor forma para definir los argumentos del constructor es usando el atributo 'index', que especifica el ubicación del argumento. De esta forma se resuelve cualquier tipo de ambigüedad.

```
<beans>
   ...

   <bean id="beanEjemplo" class="paquete.Ejemplo" >
      <constructor-arg index="0" ><ref bean="meInyectan"/></constructor-arg>
      <constructor-arg index="1" ref="meInyectanTambien"/>
      <constructor-arg index="2" type="int" value="1"/>
   </bean>

   ...
</beans>
```

7.3 Inyección por método

Se puede dar el caso, en que un bean singleton requiera como colaborador un bean no singlenton. Lo que realmente necesita el singleton, es una instancia nueva cada vez que se ejecuta alguno de sus métodos. En este caso nos encontramos con el problema de que el bean singleton es construido una única vez por el contenedor, con lo cual, siempre contendrá la misma instancia del bean no singlenton. Un forma que se nos puede plantear resolver esto es utilizando la interfaz ApplicationContextAware, y acceder al contenedor solicitando que nos retorne el colaborador. El problema de resolver esto es que atamos nuestra implementación al framework.

Para resolver esto, está la inyección de dependencias por método. Aquí, lo que hacemos es definir en el singleton un método que será el encargado de proveernos la nueva instancia del colaborador.

```
protected abstract ClaseColaborador createColaboradorNoSingleton();
```

Luego, desde el archivo de configuración de Spring, indicamos que debe reemplazar este método de forma tal de que nos genere un nuevo Colaborador del tipo ClaseColaborador cada vez que es llamado un método del singleton.

```
<bean id="unColaboradorNoSingleton"
     class="ClaseColaborador" singleton="false"/>

<bean id="unBeanSingleton" class="ClaseBeanSingleton">

   <lookup-method name="createColaboradorNoSingleton"
        bean="unColaboradorNoSingleton"/>

   <property>
   ...
   </property>

</bean>
```

Lo que hará Spring, será generar dinámicamente una clase que reescribirá el método declarado de forma tal que use al contenedor para crear la instancia necesaria del colaborador mediante el método createColaboradorNoSingleton(). Esto lo realiza utilizando la librería CGLIB que le permite generar bytecode dinámicamente.